Dead-Zone Logic in Autonomic Systems

Thaddeus Eze and Richard Anthony

Autonomic Computing Research Group
School of Computing & Mathematical Sciences (CMS)
University of Greenwich, London, United Kingdom
{T.O.Eze and R.J.Anthony}@gre.ac.uk

Abstract – **Dead-Zone logic is a mechanism to prevent autonomic managers from unnecessary, inefficient and ineffective control brevity when the system is sufficiently close to its target state. It provides a natural and powerful framework for achieving dependable self-management in autonomic systems by enabling autonomic managers to smartly carry out a change (or adapt) only when it is safe and efficient to do so –within a particular (defined) safety margin. This paper explores and evaluates the performance impact of dead-zone logic in trustworthy autonomic computing. Using two case example scenarios, we present empirical analyses that demonstrate the effectiveness of dead-zone logic in achieving stability, dependability and trustworthiness in adaptive systems. Dynamic temperature target tracking and autonomic datacentre resource request and allocation management scenarios are used. Results show that dead-zone logic can significantly enhance the trustability of autonomic systems.**

Keywords – autonomic techniques; dead-zone logic; autonomic system; trustworthiness; datacentre; dependable system; stability

I. INTRODUCTION

Autonomic Computing (AC) is now a well-established concept powered by such techniques as utility function (UF), policy based control, trend analysis (TA), and fuzzy logic (FL) etc. UFs [1] provide a means of choosing from several decision options, each expressed as a series of weighted terms. Term values represent context information and weights are reflective of system's interpretation (in terms of relevance or importance) of utility. UFs, as shown in [2], provide a natural framework for achieving self-optimisation which is a key functionality of autonomic systems (ASs). Policies are used to express overall business logic [1] that guides the decisions of the autonomic manager. TA logic identifies patterns within streams of information supplied directly from different sources (e.g., sensors, cameras, radars, RSS feed etc.). By identifying trends and patterns within a particular information stream (e.g., spikes in signal strength, fluctuation in stock prize, rising/falling trends etc.) the logic enables the autonomic manager (AM) to make more-informed control decisions and this has the potential of reducing the number of control adjustments and can improve overall efficiency and stability. Also, the analysis of recent trends enables a more accurate prediction of the future. FL provides a robust means of reasoning when decision options reflect approximate rather than fixed states. As in [3], it provides a framework for handling decisions under uncertainty and imprecision especially in a scarce model data scenario.

In terms of technical approaches, good progress has been made in AC research by using these techniques to achieve autonomic self-management [4]. Also, there have been successful efforts at combining autonomic techniques. For example, [5] presents an excellent work in combining UFs, policies, signal processing, and TA. However, current challenges include achieving trustworthy autonomic computing (TAC) and dependable ASs which the identified techniques, in isolation, are unable to fully support. The addition of Dead-zone (DZ) logic, on the other hand provides an effective means of achieving dependable systems and TAC.

TAC focuses on trustworthiness in autonomic environments. There is a careful consideration of the environmental conditions in which ASs operate. The primary concern here is not how a system operates to achieve a result but how dependable is that result from the user's perspective. The emphasis is to show that a system is capable of achieving a desired and dependable result under expected range of contexts and environmental conditions and beyond. For example, a system is not trustworthy or dependable if despite the AM making legitimate decisions within the boundaries of specified rules, there is the possibility of overall inconsistency in the behaviour of the system. This is a typical example of a situation where an AM erratically (though legally) changes its mind, thereby injecting instability into the system.

The DZ logic, first introduced in [5], is a simple mechanism to prevent unnecessary, inefficient and ineffective control brevity when the system is sufficiently close to its target state. It is implemented using a Tolerance-Range-Check (TRC) object. The TRC object encapsulates DZ logic and a three-way decision fork that flags which action amongst three options (e.g., increase value, decrease value or don't change value) to take according to the rules specified as policies. The choice of action depends on the region (i.e., upper region, in DZ, and lower region) in which the system behaviour falls. The dead-zone width, demarcated by the dead-zone boundaries, defines the behavioural bounds of the state within which a manager (AM) does not allow a system to change its action. A key use of dead-zones is to reduce oscillation and ensure stability despite high extent of adaptability.

The remainder of this paper is organised as follows: Section II looks at the background of DZ logic. Section III presents DZ logic while empirical analyses of case example scenarios are presented in Section IV. Section V concludes the work.

II. BACKGROUND

DZ logic implements a target goal (which could be dynamic) and tolerance range which is an allowable deviation from the target goal. This tolerance range defines the DZ boundary distance on either side of the target goal. For example, if the interval $l \leq T \leq u$ defines a DZ boundary (also called DZ width), then T is the target goal while l and u are the lower and upper bounds of the DZ width respectively. T is usually an average of l and u. DZ logic was introduced to AC by Anthony [5]. See Section III for details of DZ logic.

Ordinarily, achieving self-stabilisation which is an aspect of TAC could require a complex integration of different autonomic techniques. DZ logic has been shown in [1] to offer a reliable means of achieving self-stabilisation. [1] presents a mechanism to automatically monitor the stability of an autonomic component, in terms of the rate the component changes its decision (e.g., when close to a threshold tipping point). The *DecisionChangeInterval* property is implemented in the AGILE policy language [1] on decision making objects such as rules and UFs. This allows the system to monitor itself and take action if it detects instability at a higher level than the actual decision making activity. In this case, a system has to exceed a boundary by a minimum amount before action is taken. Small deviations into the dead-zone do not result in actuations.

DZ logic is used to implement the *DependabilityCheck* (DC) component of a trustworthy autonomic architecture in [6]. The DC component enables the autonomic architecture to handle longer term frame adaptation, e.g., cases where continuous validation fails to guarantee reliability. This is in the form of a longer term control that considers the behaviour of the AM over a period of time (after a certain number of decisions) to determine the effect of the AM's intervention on the system and to take corrective action if need be. Results analyses in [6] show that the implementation of DZ logic in an autonomic market scenario for adaptive just-in-time target-marketing resulted in overall gain of about 31.25% in terms of stability and cost efficiency.

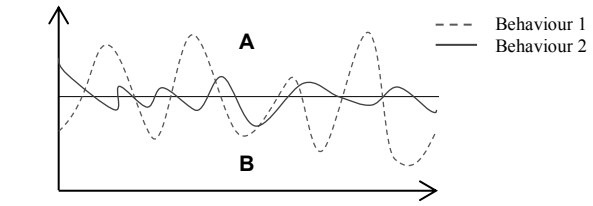
Different techniques have been used to address the issue of autonomic trustworthiness which, in our view, should be about winning the confidence of the user. Chan *et al* [7] asks the critical question of "How can we trust an AS to make the best decision?" and proposes a '*trust*' architecture to win the trust of AC system users. Also [8] in proposing an *Assurance-Driven Design* suggests that engineering design should include the detailing of a design for a solution that guarantees satisfaction of set requirements and the construction of arguments to assure users that the solution will provide the needed functionality and qualities. Any autonomic trustworthiness or dependability based technique will have to satisfy users' trust concerns that the system will remain reliable under almost all perceivable operating circumstances.

Some efforts include [9] which proposes a policy verification and validation framework that is based on model checking technique to verify the validity of administrator's specified policies in a policy-based system and [10] which proposes trustworthiness based on a fifth self-* functionality, self-regulating. The idea in [10] is that self-regulating capability is able to derive policies from high-level policies and requirements at run-time to regulate self-managing behaviours. One concern here is that proposing a fifth autonomic functionality to regulate the self-CHOP (self-Configuring, self-Healing, self-Optimising, and self-Protecting) functionalities as a solution to AS trustworthiness assumes that trustworthiness can be achieved when all four functionalities perform 'optimally'. This assumption is not entirely correct. The self-CHOP functionalities alone do not ensure trustworthiness in ASs. Take for example; the self-CHOP functionalities do not address *validation* which is a key factor in AS trustworthiness.

DZ logic provides an accessible and yet sophisticated mechanism with support for TAC. With high precision in tracking dynamic goals DZ logic is effectively suited for building high sensitive and critical systems. For example, DZ technique is well suited for building stock trading systems which track highly dynamic and fluctuating stock trends to make trading decisions. In such environment the technique will be efficient in predicting profitable and safe range within which the user can trade. This paper evaluates the impact of DZ logic on autonomics trustworthiness using two case example scenarios.

III. DEAD-ZONE LOGIC

This section discusses DZ logic in detail. Figures 1-4 provide an illustration of the use of the logic.



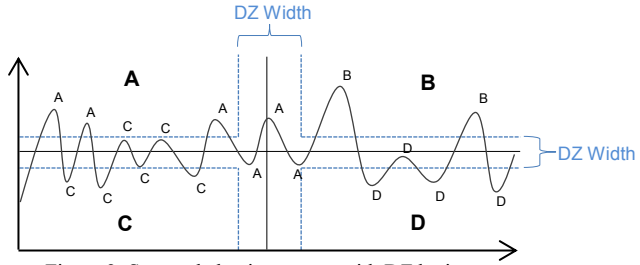


Figure 2: System behaviour zones with DZ logic

Figure 1 is the division of a system's behaviour space into two different zones (A and B) expressed in two dimensions of freedom. A particular policy action is activated within each zone. So, for example, the policy action for A is activated when the system's state falls within the boundaries of A. The two graphs represent two different behaviours of a system. For Behaviour 1 graph, action A is activated at every crest while action B is activated at every trough. There seems

to be stability in the system as the points of behaviour change (state points) are reasonably far from the zone boundary. But this cannot always be guaranteed as the system is dynamic with fluctuating behaviour. This is the case with the Behaviour 2 graph. The points of behaviour change are sometimes very close to the zone boundary causing oscillation and instability in the process.

Figure 2 introduces DZ boundaries (defining DZ widths within which a change of action is not allowed) which reduce the rate of action change and thereby increasing stability. In this case the behaviour space is divided into four zones (A-D). Without the DZ boundaries in Figure 2, there would have been 16 action changes which are now reduced to 9 by implementing DZ logic. The letters at the crests and troughs of the graph indicate which zone action is activated or running.

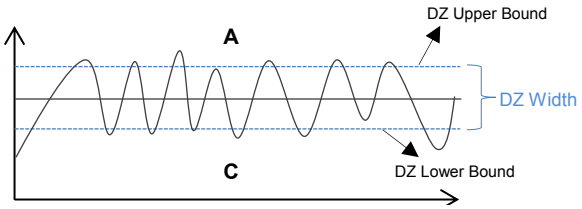


Figure 3: System behaviour with frequent adaptation.

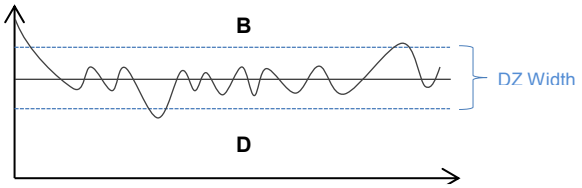


Figure 4: System behaviour with infrequent adaptation.

However, there are situations where it is necessary to dynamically adjust (tune) the DZ width. Consider Figures 3 and 4 as good examples of when it may be necessary to dynamically tune the DZ widths. In Figure 3 the system's state points are very close to the edges of the DZ boundaries. At this behaviour, it may be necessary to dynamically increase the DZ width –this is known as dynamic tuning (`DZWidth += α`). Most of the state points in Figure 4 are far from the DZ boundaries which means that the system rarely adapts so it may be necessary to reduce the DZ width (`DZWidth -= α`) if the system is desired to adapt more frequently.

The size of DZWidth, demarcated by the DZ upper and lower bounds, depends on the nature of the system and data being processed. For example, in fine-grained data instance as in the temperature controller case example scenario (Section IV A), where small shifts from the target can easily tip decisions –sometimes leading to erratic behaviour, the dead-zone boundary is expected to be small and closely tracked to the target value. However, in other cases as in the datacentre case example scenario (Section IV B), the dead-zone boundary cannot be as closely tracked to the target value. Here the target value (*DecisionBoundary*) is defined by capacity Offset and this is used by the AM to decide whether or not to deploy a server. And because Offset is populated in `serverCapacity` and depleted in `appCapacity` (i.e., the difference between available capacity and requested capacity) any behaviour shift across the decision boundary (on either side of the boundary) is in excess of `appCapacity`. This means that fluctuations around the decision boundary (the target value) are usually in multiples of `appCapacity` and to handle erratic behaviour around *DecisionBoundary* the manager will need to take `appCapacity` into consideration when calculating DZ boundaries. This explains the different boundary size calculations for the two case example scenarios. It should be noted that using improper DZ boundary size may result in over/under throttling the system or may cause the controller to have no effect on the system behaviour.

IV. CASE EXAMPLE SCENARIOS

This section considers two example scenario implementations of DZ logic. The first example is a temperature controller that tracks a dynamic goal (target temperature). The second example is a more complex system that deals with datacentre resource request and allocation management. Both experiments are simulated using C#.

A. Dynamic Temperature Target Tracking (DTTT)

Consider a room temperature (RT) controller in which, it is necessary to track a dynamic goal –a target room temperature. The target temperature (TT) is dynamic because it depends on inter alia weather condition. The AM is configured to maintain the TT by complying with the basic logic:

$$\begin{aligned} &IF\ RoomTemp < TargetTemp\ THEN\ ON\_Heating \\ &IF\ RoomTemp > TargetTemp\ THEN\ OFF\_Heating \end{aligned} \quad (1)$$

With the lag in adjusting the temperature the manager may decide to switch ON or OFF heating at every slight tick of the gauge below or above target (when RT is sufficiently close to TT). This may in turn cause oscillation which can lead to undesirable effects. In this experiment two AMs for the DTTT system are investigated. The first manager (No_DZ) does not implement DZ logic. It follows the basic logic (1) above. The second manager (With_DZ) implements DZ logic. By implementing DZ logic, With_DZ becomes sensitive to the actions of the AM. This means that it looks at the impact of the AM's action on the system over a longer term time frame and decides whether to retune itself. In this case, for example, if the actions of the AM cause the system to oscillate (e.g., frequently switching heating), With_DZ creates a tolerance behaviour range within which the AM is not allowed to change its actions –i.e., actions are persisted. Firstly, the upper and lower bounds are calculated as follows:

$$\begin{aligned} DZUpperBound &= oldDZUpperBound \pm DZConst \\ DZLowerBound &= oldDZLowerBound \mp DZConst \end{aligned} \quad (2)$$

Where *DZConst* is a tuning parameter used to adjust the DZWidth. This will calm the system's erratic behaviour. However, if the erratic behaviour does not drop to an acceptable level the manager can further retune itself by increasing the DZ boundary –i.e., rerunning (2) with '+' for the upper bound and '–' for the lower bound. If on the other hand the manager discovers that it is not making decisions frequently enough (i.e., the room is getting too cold or too hot), it can retune its behaviour to increase its rate of decision-making by reducing the DZ boundary –i.e., rerunning (2) with '–' for the upper bound and '+' for the lower bound. So the manager retunes itself by dynamically adjusting the DZ boundary by running (2) as appropriate.

- **Results**

The main focus in this analysis is to investigate the impact of DZ logic in an autonomic temperature controller. Figures 5-8 are representation of the system behaviour. In this experiment the target temperature (TT) is set at 20^0C and the room temperature (RT) fluctuates between 15^0C and 25^0C. The algorithm used in generating the RT is modeled to replicate real life RT pattern according to the study in [11]. The No_DZ manager tracks the TT by implementing the basic logic in (1). This results in the manager frequently and inefficiently changing its decision as shown by the *Heating_NoDZ* graph. The With_DZ manager implements (1) with DZ logic to calm the expected fluctuation. At first, DZ is implemented without a dynamic DZ boundary tracking (Figure 6). For example, at points (A) and (B), where RT moved below and beyond TT, No_DZ switched heating ON and OFF respectively while With_DZ did not change its actions as both points were still within the DZ boundary.

In Figure 5 the With_DZ manager is configured to dynamically adjust the DZ boundary for efficient tracking of TT. This is achieved by continuously taking a longer term view (at interval of 10 decision cycles) of the system and implementing **Rule 1**. A cycle is an instance of RT generation and control decisions are made at every cycle.

The DZUpperBound and DZLowerBound (in Figure 5) are dynamically adapted (following **Rule 1**) to calm the behaviour of the With_DZ manager. The manager becomes more sensitive to fluctuations in RT and thus efficiently tracks the TT as shown in the trend analysis graph (Figure 8). This also results in high stability for With_DZ.

The *Heating_NoDZ* and *Heating_WithDZ* graphs represent the behaviour trends (in terms of turning heating ON or OFF) of the two AMs. These trends show the state of heating as controlled by the two AMs. Crests indicate that heating is turned ON while troughs indicate that heating is turned OFF. The AM action trends in Figures 5 and 6 indicate that No_DZ has high level of fluctuation between decisions – i.e., between ON and OFF states. This is mirrored in Figure 7.

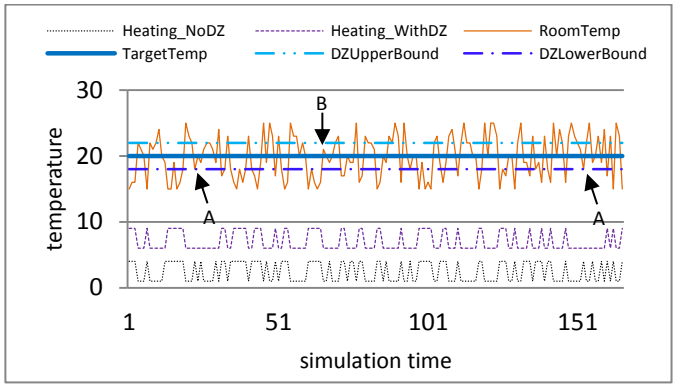


Figure 6: System behaviour analysis. DZ boundary is static

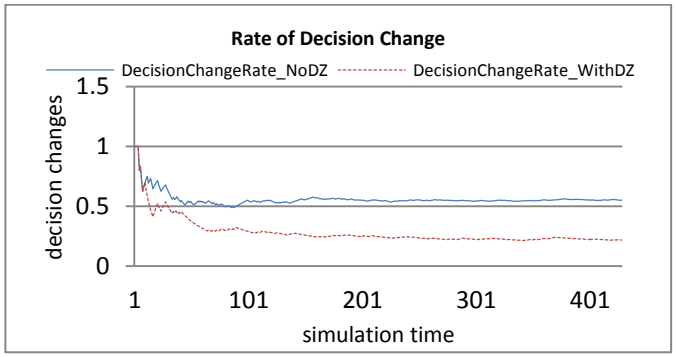


Figure 7: Rate of decision change analysis

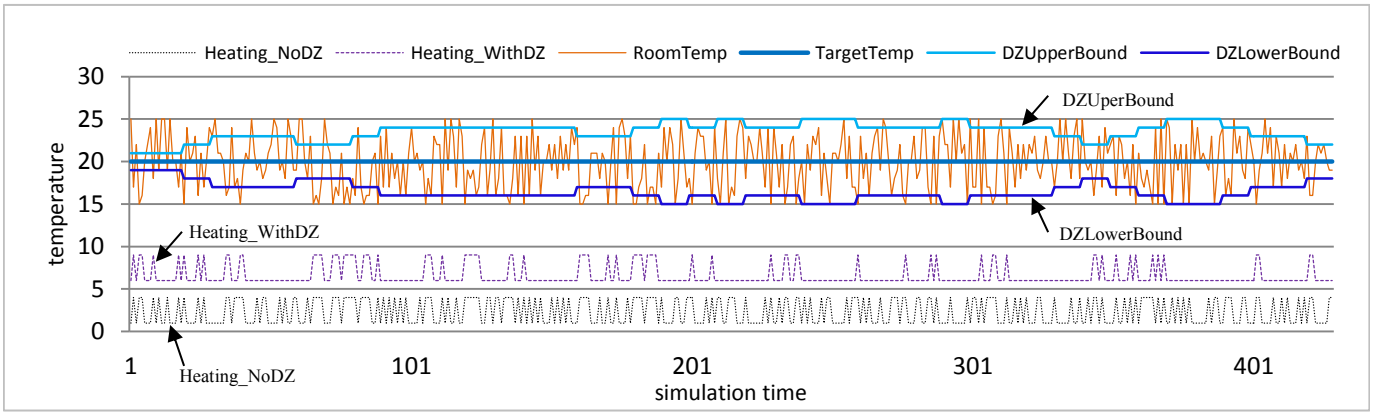


Figure 5: System behaviour analysis. DZ boundary is dynamically tracked

```
Rule 1: (repeat rule every 10 decision cycles)
  increase DZWidth by 1 if DecisionChangeCount >= 4
decrease DZWidth by 1 if DecisionChangeCount < 1
```

Results show that while No_DZ changed its decision a total of 236 times, With_DZ changed its decision 93 times. This represents a significant improvement with DZ logic. Imagine a stock trading system making 236 trading decisions whereas it can make only 93 efficient decisions within the same time frame and conditions by simply implementing DZ logic. The trend analysis graph (Figure 8) illustrates the level of efficiency of each manager.

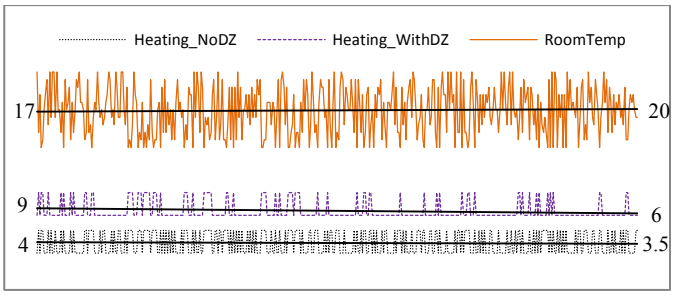


Figure 8: Trend analysis of manager behavior. Note that graph is on same scale as Figure 5.

Figure 8 is a replica of Figure 5 with additional TrendLine to each of the graphs. Although values shown are on the temperature scale, what is important is the magnitude of fluctuation in the observed trends. As shown, the trend of RT moved from 17 to 20. To adequately adapt the RT, a heating trend (HT) of the same magnitude (e.g., 23 to 20) in the opposite direction is required. As observed, With_DZ HT moved in the opposite direction of RT trend with the same magnitude (9 to 6). If we superimpose With_DZ HT over the RT trend and take average across the trend, the RT will be maintained at approximately 20^0C which is the TT. With the No_DZ HT, RT would be maintained at approximately 17.25^0C (i.e., 34.5/2) which is well below the TT.

B. Datacentre Resource Request and Allocation Management

The experimental analysis in this case example scenario investigates the performance impact of DZ logic on a datacentre AM. The purpose of this simulation is not to investigate datacentres but to analyse the performance of two AMs in a datacentre resource allocation and management scenario. The work here does not propose any new scheduling algorithm for efficient utilisation of datacentre resources; however it uses basic resource allocation technique to model the performance of datacentre AMs in terms of the effectiveness of resource request and allocation management. Other research, e.g., [12] and [13] have proposed scheduling algorithms that optimise the performance of datacentres.

The datacentre model used in this simulation comprises a pool of resources S_i (servers), a list of applications A_j, a pool of services $\underset{\sim}{U}$ (a combination of applications and their provisioning servers), and an AM that optimises the entire system. A_j and S_i are, respectively, a collection of applications supported (as services) by the datacentre and a collection of servers available to the AM for provisioning (or scheduling) available services according to request. The AM dynamically populates $\underset{\sim}{U}$ to service arriving requests. $\underset{\sim}{U}$ is defined by (3):

$$\underset{\sim}{U} = \begin{cases} A_1\text{: } (S_{11}, S_{12}, S_{13}, \ldots, S_{1i}) \\ A_2\text{: } (S_{21}, S_{22}, S_{23}, \ldots, S_{2i}) \\ \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ A_n\text{: } (S_{n1}, S_{n2}, S_{n3}, \ldots, S_{ni}) \end{cases} \tag{3}$$

Where A_i: (S_i ... S_n) means that (S_i ... S_n) servers are currently allocated to Application A_i and n is the number of application entries into $\underset{\sim}{U}$.

Service (application) requests are queued. Individual requests are served only if there are enough resources otherwise they remain in the queue (or may eventually be dropped). The manager checks for resource availability and deploys server(s) according to the size of the request. The size of application requests and the capacity of servers are defined in million instructions per second (MIPS). A deployed server is first placed in a queue for a time defined by the variable *ProvisioningTime*. This queue simulates the time (delay) it takes to load or configure a server with necessary application.

- **Manager Logic**

This explains the basic logical composition of the AMs used. Two AMs, represented by SysA and SysB are analysed. SysA does not implement DZ logic while SysB does.

SysA:

This AM implements the basic autonomic control logic based on the traditional MAPE (Monitor, Analyse, Plan, and Execute) logic [14]. The manager receives requests and allocates resources according the explained scheduling algorithm. The basic allocation logic here is to deploy a server whenever capacity offset (i.e., excess capacity of running servers –these are used to service new requests) is less than the current capacity of a single request. This is known as the *DecisionBoundary*. This is depicted, for example, as:

```
if (appOffset < appCapacity)
  { <...deploy server...> }
```

Where

```
appOffset = appAvailableCapacity - appRunningCapacity;
```

The manager does not start a job that cannot be completed –i.e., at every *DecisionBoundary* the AM checks to make sure that it has enough resources to service a request otherwise it rejects the request and updates itself. However, the AM does not consider the rate at which system behaviour crosses the *DecisionBoundary* which is addressed by SysB.

SysB:

SysB performs all the activities of the SysA with additional intelligence. The manager looks at the balance of cost over longer term and retunes its configuration to ensure a balanced performance. This is achieved by implementing DZ logic on decision boundaries. Firstly, the DZ boundaries (upper and lower bounds), for example, are calculated as:

```
DZUpperBound = (appCapacity + (appCapacity * DZConst));
DZLowerBound = (appCapacity - (appCapacity * DZConst));
```
(4)

Secondly, the zone areas are defined as follows (two zones are defined with one on either side of the *DecisionBoundary* –see Figure 9):

```
if (appOffset < appCapacity)
        {SystemBehaviour = "IsInDeployZone"; }
else
       {SystemBehaviour = "IsNotInDeployZone"; }
```

Then stability is maintained by persisting the behaviour (DecisionBoundary outcome) of the system across the zones as follows:

```
if (appOffset >= appCapacity)
       {SystemBehaviour = "IsNotInDeployZone"; }
if ((SystemBehaviour == "IsInDeployZone") && (appOffset
      <DZUpperBound))
       {SystemBehaviour = "IsInDeployZone"; }
else
       {SystemBehaviour = "IsNotInDeployZone"; }
if ((SystemBehaviour == "IsNotInDeployZone") &&
      (appOffset > DZLowerBound))
       {SystemBehaviour = "IsNotInDeployZone"; }
else
       {SystemBehaviour = "IsInDeployZone"; }
```

Thus the *DecisionBoundary* in SysA which is `(appOffset < appCapacity)` now becomes `(SystemBehaviour == "IsInDeployZone")` in SysB. The AM dynamically changes the `DZConst` value between three values of 1, 1.5 and 2. By doing this the AM is sensitive to its own behaviour and thus proactively regulates (retunes) its decision pattern to maintain stability and reliability.

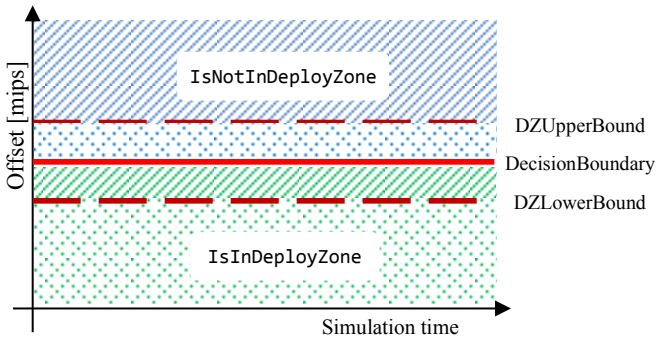


Figure 9: Dead-zone logic implemented by SysB.

In Figure 9 the area shaded in green represents the '*IsInDeployZone*' which means the manager should deploy a server while the area shaded in blue represents the '*IsNotInDeployZone*' which means the manager should not deploy a server. Likewise, the dotted shade pattern represents the '*IsInDeployZone*' while the diagonal shade pattern represents the '*IsNotInDeployZone*'. As shown, if, for example, the system behaviour falls within the '*IsNotInDeployZone*' area, the manager persists the action associated to this zone until system behaviour falls below the '*DZLowerBound*' boundary at which point the action associated to the '*IsInDeployZone*' area is activated. This way the AM is able to maintain reliability and efficiency. The AM also retunes its behaviour (as explained earlier) by adjusting *DZWidth* if fluctuation is not reduced to an acceptable level. Thus, three behaviour regions (in which different actions are activated) are defined; 'upper region' (*IsNotInDeployZone* with 'DO NOT DEPLOY SERVER' action), 'lower region' (*IsInDeployZone* with 'DEPLOY SERVER' action), and 'in DZ' (within the DZWidth with either of the two actions). It is important to note, as shown in Figure 9, that within the DZ boundary (i.e., the 'in DZ' region), either of the actions associated to '*IsInDeployZone*' and '*IsNotInDeployZone*' areas could be maintained depending on the 'current action' prior to deviation into the 'in DZ' region. So actions activated in the 'upper region' and 'lower region' are respectively persisted in the 'in DZ' region. This is further explained in Figure 10 which shows the resultant effect of the DZ logic in terms of what zone action is activated per time.

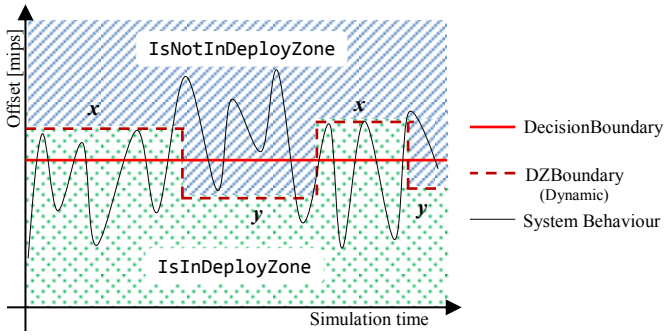


Figure 10: Illustration explaining actual performance effect of DZ logic.

Figure 10 explains what happens in Figure 9. As system behaviour fluctuates around decision boundary, the manager dynamically adjusts the DZ boundary to mitigate erratic adaptation. As shown, minor deviations across the DecisionBoundary do not result in decision (or action) change. In this case (Figure 10) actions for `IsInDeployZone` and `IsNotInDeployZone` are persisted at states *x* and *y* respectively despite system behaviour crossing the decision boundary at those state points.

- **Results**

Table 1 is a collection of the major parameters used in this experimental analysis.

Table 1: Simulation parameters

Parameter	Value
# of servers	300
# of applications	4
Request rate	1 req/sec
Application capacity (MIPS)	20000
Server capacity (MIPS)	40000
Managers	SysA and SysB

In this simulation, there are 300 servers of 40000 MIPS capacity each. This means there is a total of 12000000 MIPS to share between requests for four applications. Reclaimed servers are later added to this available capacity. If the total requested capacity is higher than the total provisioned capacity, the unused server list will be empty (leaving the manager with a deficit of outstanding requests without resources to service them) and the datacentre is overloaded. Results show that while SysA deployed about 303 servers to serve 586 requests, SysB deployed 269 servers to serve 569 requests. At the end of the simulation SysA had 3 outstanding (unused) servers while SysB had 36. Figures 11 and 12 give a breakdown of the AMs performances.

The difference between requested capacity and provisioned capacity (or in real time analysis, running capacity and available capacity) is known as *Offset*. Where offset is close to zero, the difference with respect to running and available MIPS is low and the manager is therefore very efficient. When offset is much greater than or much less than zero, the manager is over-provisioning or under-provisioning respectively and is very inefficient. The AMs are designed to have a window of 'optimum provisioning' defined by the interval ($0 \leq Offset \leq AvgAppCapacity$) which means that the managers are configured to maintain *AvailableCapacity* of up to average *appCapacity* for just-in-time provisioning. However, AM efficiency is defined by its ability to maintain *Offset* as close as possible to zero. Figure 11 shows the efficiency analysis of the AMs in terms of maximising resources.

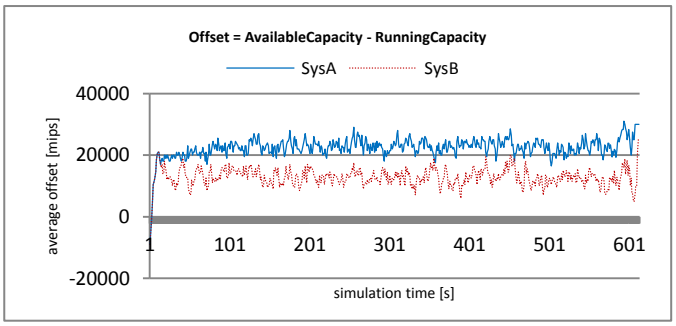


Figure 11: Manager efficiency analysis

Figure 11 shows that, in terms of efficiency, SysB significantly outperforms SysA. Though SysA checks to ensure resource availability against resource requests, it is not adequately sensitive to erratic request fluctuation. High level of erratic request fluctuation disorientates SysA but this effect is naturally and dynamically handled by SysB by implementing DZ logic. SysB takes a longer term look at the self-management effect on the datacentre and retunes its self-management behaviour. The rate at which the AMs change decision (which can indicate erratic behaviour) is indicated by the gap between the crests and troughs of the graph in Figure 11. Smaller gap indicates erratic change of decision while bigger gap indicates more persisted decision. As clearly seen, SysB has significantly more persisted decisions and this allows it to more adequately track resource availability against resource requests which leads to more efficient performance. Recall that optimum provisioning is defined by the ($0 \leq Offset \leq AvgAppCapacity$) interval which in this case is between 0 and 20000 MIPS. sysDC clearly falls within this range, though a bit towards the 20000 border (Figure 11). This means that while SysA tries to maintain AvailableCapacity of up to 20000 MIPS for just-in-time provisioning, SysB efficiently depletes this reserve to maximise resources while at the same time maintaining the same level of performance and even better compared to SysA. This is replicated in the deployment rate analysis (Figure 12).

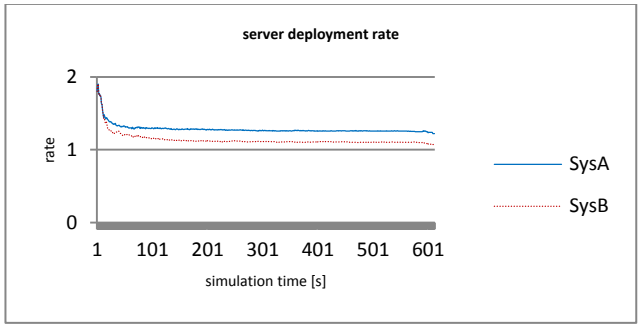


Figure 12: Server deployment rate analysis

Figure 12 shows the rate at which the AMs deploy servers as requests arrive. With the same rate of request arrival, SysA deployed the most servers than SysB. This is because SysA follows a strict decision boundary which allows it to change decisions as system behaviour fluctuates around the decision boundary (Figure 9). This explains why SysA easily runs out of servers while SysB still retains a couple of unused servers because SysB implements DZ logic around the decision boundary. By implementing DZ logic SysB keeps performance very close to the optimum mark which indicates high efficiency.

V. CONCLUSION

We have presented empirical analyses that demonstrate the effectiveness of dead-zone (DZ) logic in achieving stability, dependability and trustworthiness in autonomic computing. Analyses show that DZ logic is sufficiently sophisticated in building autonomic systems (ASs) that can operate efficiently and yield satisfactory results under almost all perceivable operating circumstances. This has the capability of extending the trustability of ASs. Ordinarily, achieving self-stabilisation which is an aspect of trustworthy autonomic computing (TAC) could require a complex integration of different autonomic techniques. DZ logic has

been shown to offer a reliable means of achieving self-stabilisation, dependable systems and TAC.

REFERENCES

[1] R. Anthony, "Policy-based autonomic computing with integral support for self-stabilisation," *in International Journal of Autonomic Computing*, Vol.1, No.1, 2009, pp.1-33.

[2] W. Walsh, G. Tesauro, J. Kephart, and R. Das, "Utility Functions in Autonomic Systems", *in Proceedings of the first International Conference on Autonomic Computing (ICAC)*, 2004, New York, USA.

[3] A. Andrzejak, S. Graupner and S. Plantikow, "Predicting Resource Demand in Dynamic Utility Computing Environments," *in Proceedings of the second International Conference on Autonomic and Autonomous Systems (ICAS),* 2006, California, USA.

[4] T. Eze, R. Anthony, C. Walshaw, and A. Soper, "Autonomic Computing in the First Decade: Trends and Direction," *in Proceedings of the Eighth International Conference on Autonomic and Autonomous Systems (ICAS)* 2012, St. Maarten, Netherlands Antilles.

[5] R. Anthony, "Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques," *in Proceedings of the fourth International Conference on Autonomic Computing (ICAC), 2007*, Florida, USA.

[6] T. Eze, R. Anthony, C. Walshaw, and A. Soper, "A New Architecture for Trustworthy Autonomic Systems", *in Proceedings of the Eighth International Conference on Autonomic and Autonomous Systems (ICAS)* 2012, St. Maarten, Netherlands Antilles.

[7] H. Chan, A. Segal, B. Arnold, and I. Whalley, "How Can We Trust an Autonomic System to Make the Best Decision?" *in Proceedings of the second International Conference on Autonomic Computing (ICAC), 2005*, Seattle, USA.

[8] J. Hall and L. Rapanotti, "Assurance-driven design in Problem Oriented Engineering," *in International Journal On Advances in Intelligent Systems (IntSys)*, volume 2, number 1, 2009, pp. 26-37.

[9] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, "Constraint Verification for Concurrent System Management Workflows Sharing Resources," *in Proceedings of the third International Conference on Autonomic and Autonomous Systems (ICAS), 2007*, Athens, Greece.

[10] X. Li, H. Kang, P. Harrington, and J. Thomas, "Autonomic and trusted computing paradigms," *in Lecture Notes in Computer Science*, Volume 4158, 2006, pp. 143-152.

[11] A. Melikov, U. Krüger, G. Zhou, T. Madsen, and G. Langkilde, "Air Temperature Fluctuations in Rooms", *in Building and Environmental Journal*, Vol 32, Issue 2, March 1997, pp. 101–114

[12] R. Das, J. Kephart, J. Lenchner, and H. Hamann, "Utility-function-driven energy-efficient cooling in data centers," *in Proceeding of the Seventh International Conference on Autonomic Computing (ICAC)*, 2010, New York, USA.

[13] I. Goiri, J. Fit´o, F. Juli`a, R. Nou, J. Berral, J. Guitart and J. Torres, "Multifaceted Resource Management for Dealing with Heterogeneous Workloads in Virtualized Data Centers*," in Proceedings of Eleventh IEEE/ACM International Conference on Grid Computing* (GRID), 2010, Brussels, Belgium.

[14] J. Kephart and D. Chess, "The Vision of Autonomic Computing," Computer, vol. 36, no. 1, pp. 41-50, 2003.